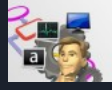
Ajustar a página

Análisis de algoritmos

Tema 02: Divide y vencerás (DyV)

Contenido

- Introducción
- Divide y vencerás
- Complejidad de divide y vencerás
  - Ejemplo 01: Búsqueda del máximo y del mínimo
  - Ejemplo 02: Ordenamiento de un arreglo
- Observaciones sobre Divide y Vencerás

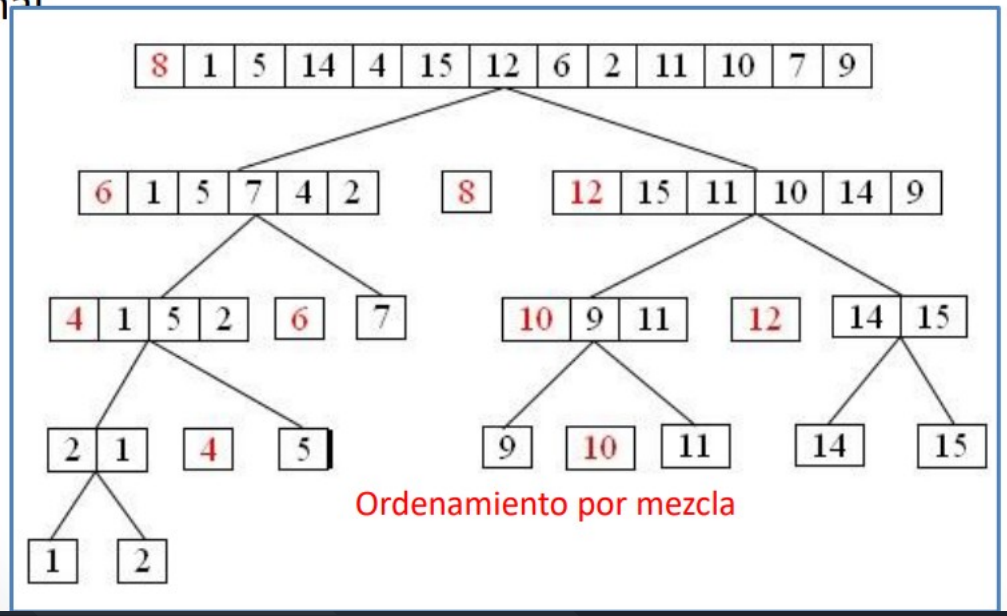


Introducción

- En la cultura popular, divide y vencerás hace referencia a un refrán que implica resolver un problema difícil, dividiéndolo en partes más simples tantas veces como sea necesario, hasta que la resolución de las partes se torna obvia. La solución del problema principal se construye con las soluciones encontradas.

- En las ciencias de la computación, el término divide y vencerás hace referencia a uno de los más importantes paradigmas de diseño algorítmico.

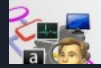
- El paradigma divide y vencerás, está basado en buscar la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar (Sin que se solapen). El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original



Ordenamiento por mezcla

- **Dividir y vencer** es la base de varios **algoritmos eficientes** para casi cualquier tipo de problema como, por ejemplo, algoritmos de ordenamiento *(mergesort, quicksort, entre otros),* multiplicar números grandes *(Karatsuba),* análisis sintácticos *(top-down),* la transformada discreta de Fourier, multiplicación rápida de matrices (*Strassen),* etc.

- **Analizar y diseñar algoritmos de DyV** son tareas que lleva tiempo dominar. Al igual que en la inducción, a veces es necesario sustituir el problema original por uno más complejo para conseguir realizar la recursión, y **no hay un método sistemático de generalización**.



Divide y vencerás

- "El método divide y vencerás consiste en descomponer el problema en una serie de subproblemas de menor tamaño, al resolver estos subproblemas usando siempre la misma técnica, las soluciones parciales se combinan para obtener la solución del problema original".

- *"Tenemos un problema complejo al cual dividimos en subproblemas mas pequeños a resolver. Para resolver cada subproblema seguimos el mismo procedimiento hasta que llegamos a un problema que es trivial. Una vez resueltos los subproblemas los combinamos para dar solución al problema original".*

- De esta forma **DyV se expresa de manera natural mediante un algoritmo recursivo.**

```
fun divide_y_venceras(x: problema) dev y:solución
    si pequeño(x) entonces
        y:=metodo_directo(x)
    si no
        {descomponer x en k >= 1 problemas más pequeños}
        {x1, x2, ..., kx}:=descomponer(x)}
        {resolver recursivamente los subproblemas}
        para j=1 hasta k hacer
            yj:= divide_y_venceras(xj)
        fin para
        {combinar los yj para obtener una solución y para x}
        y:=combinar(y1, ..., yk)
    fin si
fin
```

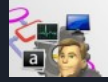
La función pequeño(x) es un predicado que determina si el tamaño del problema x es suficientemente pequeño para ser resuelto sin dividir más. Si es ese el caso, el problema se resuelve mediante la función método_directo(x).

- Para que la aplicación del método divide y vencerás, convenga debe cumplirse que:
  1. Las operaciones descomponer y combinar deben ser bastante eficientes.
  2. El número de subproblemas generados sea pequeño
  3. Los subproblemas sean aproximadamente del mismo tamaño y no solapen entre sí.

- La función complejidad para un problema de tamaño *n* es un sistema de ecuaciones recurrentes de la forma:

$$f(m) = f_{CalculaSoluciónInmediata}(m)$$

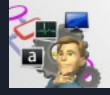
$$f(n) = f_{Divide}(n) + f_{Combina}(n) + \sum_{i=1}^{k} f(t[i])$$



- Si el problema x es de tamaño n, y los tamaños de los subproblemas $x_1, x_2, \dots, x_k$ son, respectivamente, $n_1, n_2, \dots, n_k$, podemos describir el coste en tiempo del algoritmo ***divide_y_venceras*** mediante la siguiente recurrencia:

$$T(n) = \begin{cases} g(n), & n \leq n_0 \\ \sum_{j=1}^{k} T(n_j) + f(n), & n > n_0 \end{cases}$$

- Donde $\boldsymbol{T(n)}$ es el tiempo del algoritmo **divide_y_venceras**, $\boldsymbol{f(n)}$ es el tiempo que toma **combinar** las soluciones y $\boldsymbol{g(n)}$ es el tiempo del **metodo_directo**.

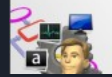


Complejidad de divide y vencerás

- El diseño **Divide y Vencerás produce algoritmos recursivos** cuyo tiempo de ejecución se puede expresar mediante una ecuación en recurrencia del tipo:

$$T(n) = \begin{cases} cn^k, & si\ 1 \leq n < b \\ aT(n/b) + cn^k, & si\ n \geq b \end{cases}$$

- $\boldsymbol{a, c\ y\ k}$ son números reales
- $\boldsymbol{n}$ **y** $\boldsymbol{b}$ son números naturales,
- $\boldsymbol{a > 0,\ c > 0,\ k \geq 0\ y\ b > 1}$
- $\boldsymbol{a}$ representa el número de subproblemas.
- $\boldsymbol{n/b}$ es el tamaño de cada uno de ellos.
- $\boldsymbol{cn^k}$ representa el coste de descomponer el problema inicial en los a subproblemas y el de combinar las soluciones para producir la solución del problema original, o bien el de resolver un problema elemental.

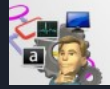


- La solución a esta ecuación, puede alcanzar distintas complejidades.

$$T(n) = \begin{cases} \Theta(n^k) & si\ a < b^k \\ \Theta(n^k \log n) & si\ a = b^k \\ \Theta(n^{log_b a}) & si\ a > b^k \end{cases}$$

- Las diferencias surgen de los distintos valores que pueden tomar a y b, que en definitiva determinan el número de subproblemas y su tamaño. Lo importante es observar que en todos los casos la complejidad es de orden polinómico o polilogarítmico pero nunca exponencial.

- Los algoritmos recursivos pueden alcanzar una complejidad exponencial en muchos casos. Esto se debe normalmente a la repetición de los cálculos que se produce al existir **solapamiento en los subproblemas** en los que se descompone el problema original.



Ejemplo 01: Búsqueda del máximo y del mínimo

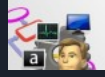
Método directo

```
MaxMin (A: array [1..N] of tipo; var Max, Min: tipo)
        Max = A[1]
        Min = A[1]
        para i=2, 3, ..., N
                si A[i]>Max
                        Max = A[i]
                en otro caso si A[i]<Min
                        Min = A[i]
```

Operación básica: Asignaciones a Max y Min.

Complejidad temporal

Peor caso: Los números están hasta el final el máximo y el mínimo. $Ft(n) = n + 1 \in O(n)$



Divide y vencerás

```
MaxMinDV (i, j: integer; var Max, Min: tipo)
        si i<j-1
        /*Dividir el problema en 2 subproblemas*/
                mit = (i+j) div 2
                MaxMinDV (i, mit, Max1, Min1)
                MaxMinDV (mit+1, j, Max2, Min2)
        /*Combinar*/
                si Max1>Max2
                                Max= Max1
                en otro caso
                                Max = Max2
                si Min1<Min2
                        Min = Min1
                en otro caso
                        Min = Min2
        /*Caso base*/
        en otro caso si i=j-1
                si A[i]>A[j]
                        Max = A[i] ; Min = A[j]
                en otro caso
                        Max = A[j] ; Min= A[i]
        en otro caso
                Max = A[i] ; Min = Max
```

Operación básica: Asignaciones a Max, Min, Max1, Min1, Max2 y Min2.

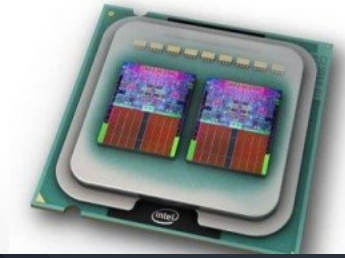
Complejidad temporal

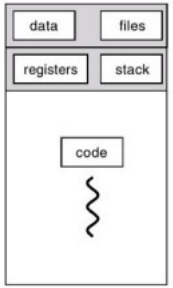
Peor caso: Los números están hasta el final el máximo y el mínimo.

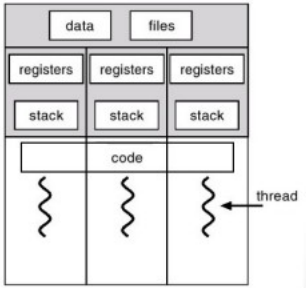
$$T(n) = \begin{cases} 2, & si\ 1 \leq n \leq 2 \\ 2T(n/2) + 2, & si\ n > 2 \end{cases}$$

$$T(n) \in O(n)$$

- En el ejemplo anterior la complejidad temporal no obtuvo un beneficio al utilizar un algoritmo que emplea la técnica de DyV, si se piensa como una aplicación a funcionar bajo la idea de un proceso **monoprocesador,** pero si se considera que la solución de los **problemas parciales es independiente** y se utiliza una **idea de procesamiento paralelo**, probablemente **el potencial de un algoritmo** como este se vera reflejado en los tiempos de procesamiento de problemas para **"n" muy grandes**.







Ejemplo 02: Ordenamiento de un arreglo

Método directo *(Ej. Inserción)*

```
Procedimiento Insercion(A,n)
{
        para i=1 hasta i<n hacer
                temp=A[i]
                j=i-1
                mientras((A[j]>temp)&&(j>=0)) hacer
                        A[j+1]=A[j]
                        j--
                fin mientras
                A[j+1]=temp
        fin para
fin Procedimiento
```

Operación básica: Movimientos en el arreglo y comparaciones entre un elemento y "temp"

Complejidad temporal

Peor caso: Los números están ordenados y se compara un elemento con todos los elementos del arreglo . $Ft(n) = n * (n-1) \in O(n^2)$

Divide y vencerás (Mergesort)

```
void mergeSort(int *arr, int size)
{
        /*Caso base*/
    if (size == 1)
        return;

        /*Dividir el problema en 2 subproblemas*/
        int size1 = size/2, size2 = size-size1;
        mergeSort(arr, size1);
    mergeSort(arr+size1, size2);

        /*Combinar*/
        merge(arr, size1, size2);
}
```

Complejidad temporal

$$T(n) = \begin{cases} 1, & si\ 1 = 1 \\ 2T(n/2) + n, si\ n > 2 \end{cases}$$

$$T(n) \in O(n \log n)$$

6 5 3 1 8 7 2 4

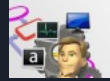
Observaciones sobre Divide y Vencerás

1. En primer lugar, **el número *k* debe ser pequeño e independiente de una entrada determinada**.

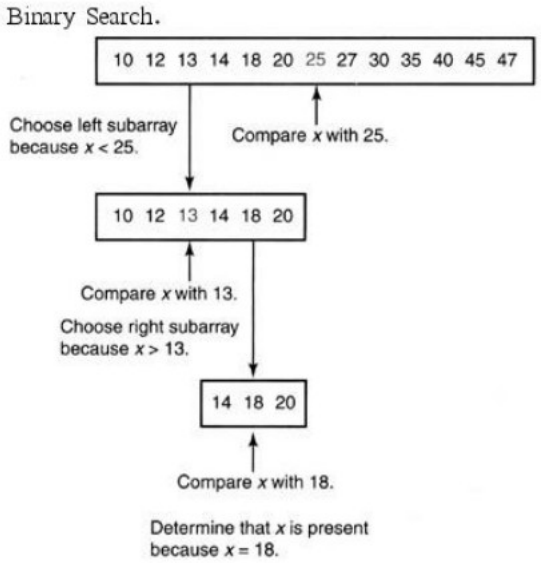
   - En el caso particular de los algoritmos Divide y Vencerás que contienen **sólo una llamada recursiva**, es decir ***k*=1**, hablamos de **algoritmos de *simplificación***.

     - Tal es el caso del algoritmo recursivo que resuelve el cálculo del **factorial de un número**, que sencillamente reduce el problema a otro subproblema del mismo tipo de tamaño más pequeño.

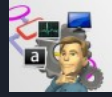
     - También es un algoritmo de simplificación el de **búsqueda binaria en un vector.**



- El algoritmo de búsqueda binaria es un ejemplo de un algoritmo de simplificación, pero posee la característica de dividir el problema para alcanzar la solución de una manera más simple.

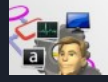


Búsqueda binaria



2. Desde un punto de vista de la eficiencia de los algoritmos **Divide y Vencerás**, es muy importante conseguir que **los subproblemas son independientes, es decir, que no existe solapamiento** entre ellos.

- Si la solución de los subproblemas no es independiente el tiempo de ejecución de estos algoritmos será exponencial.
  - Tal es el caso la serie de Fibonacci, la cual, a pesar de ajustarse al esquema general y de tener sólo dos llamadas recursivas, tan sólo se puede considerar un algoritmo recursivo pero **no clasificarlo** como diseño Divide y Vencerás.



3. Debe haber **reparto de carga**, es decir, **la división de los subproblemas** debe ser lo más **equilibrada** posible.
   - Si la solución de los subproblemas no tiene un coste similar para todos puede pasar que el coste caiga en un coste poco beneficioso que si se resolviera con el algoritmo clásico.
     - Tal es el caso del algoritmo de ordenación rápida, si al realizar la partición del vector nos encontramos con que el pivote queda casi al principio o al final del vector, tendremos que el tiempo de ejecución del algoritmo está en $O(n^2)$, mientras que si esta próximo a la mitad del vector, estará en $O(n \log(n))$.